# Preface:

* Spring Applications, complexity increases with addition of more features

**Problems usually faced:**

* Configuration is getting repeated for complex applications.
* Need to add a lot of dependencies to the servlet XML.
* Configuring Spring applications tend to become challenging and error-prone, which includes several steps in set-up and configuration, build and deploy steps.



# Introduction to Spring Boot

Spring Boot allows you to build stand-alone, operating, production-grade Spring based Applications with ease.

The guiding principle of Boot is **convention over configuration**.

Let’s have a look at some of the important features in Boot:

* [**Starter modules**](#_Spring_Boot_Starters) for simplifying dependency configuration
* [**Auto-configuration**](#_Spring_Boot_Auto-Configuration) whenever possible
* [**Advanced externalized configuration**](#_Advanced_externalized_configuration)
* Embedded, built-in Tomcat, Jetty or Undertow
* **Stand-alone** Spring applications
* [**Production-ready features**](#_Add_production-grade_services) such as metrics, health checks, and externalized configuration
* No requirement for XML configuration

# Spring Boot Starters

Simply put, starters are dependency descriptors that reference a list of libraries.

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.0.2.RELEASE</version>  <relativePath />  </parent> |

* You only need to specify the dependency version once for the parent.
* It eliminates potential errors related to incompatible library versions.
* When you need to update the Boot version, you only need to change a single, central version, and everything else gets implicitly updated.

**https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-starter**

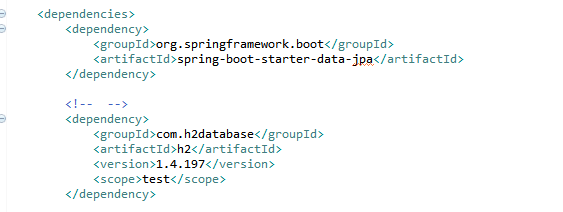
# **Spring Boot Auto-Configuration**

* Spring Boot aims to simplify this process by providing a **sensible default configuration, based on the dependencies on the classpath** and loaded automatically behind the scenes.
* It’s important to understand that, as soon as you define **your configuration beans**, then these **will take precedence over the auto-configured ones**.

**Example**:

To work with Spring Data JPA, we also need to set up a database.

Luckily, Boot provides auto-configuration for three types of in-memory databases: H2, HSQL, and Apache Derby.



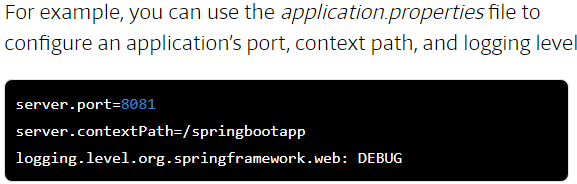
**Auto-configures Hibernate as the default JPA provider.**

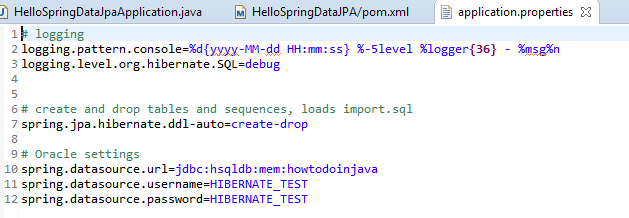
# Advanced externalized configuration

Configure the behavior of an application via:

**External properties files | YAML files | Environment variables | Command-line arguments.**

These properties have standard names that will be automatically picked up by Boot and evaluated in a set order.



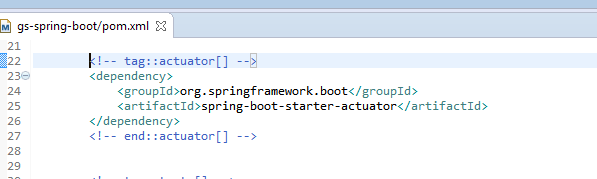


<https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html>

# **Add production-grade services**

**If you are building a web site for your business, you probably need to add some management services. Spring Boot provides several out of the box with its actuator module, such as health, audits, beans, and more.**

**Add this to your build file’s list of dependencies:**



You will see a new set of RESTful end points added to the application. These are management services provided by Spring Boot

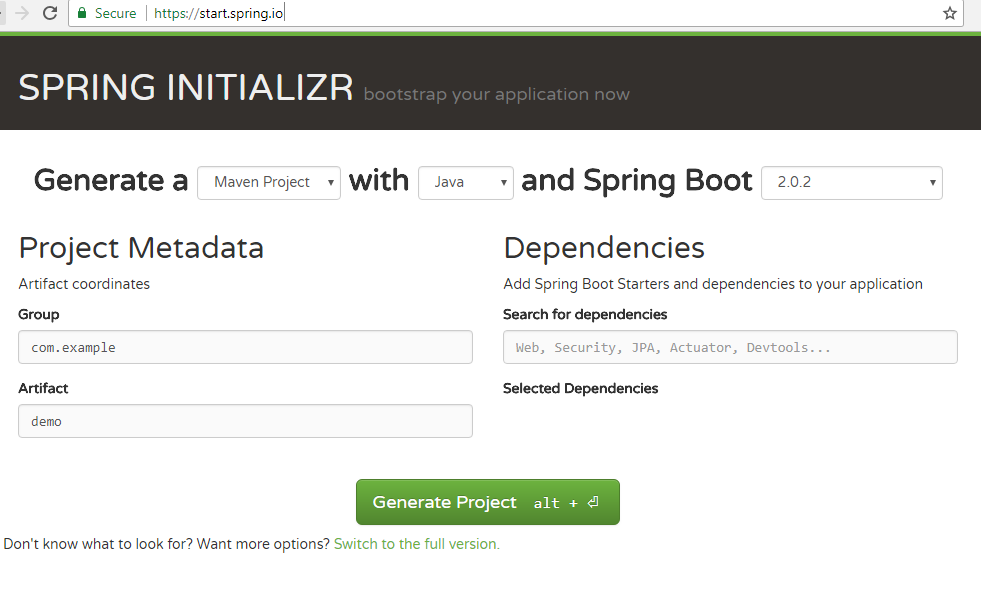
They include: errors, [actuator/health](http://localhost:8080/actuator/health), [actuator/info](http://localhost:8080/actuator/info), [actuator](http://localhost:8080/actuator).

<https://docs.spring.io/spring-boot/docs/2.0.2.RELEASE/reference/htmlsingle/#production-ready-endpoints>

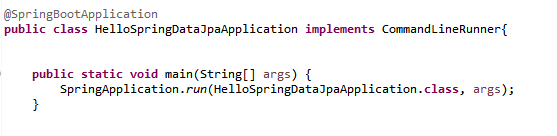
# **Spring Boot Initializr**

Use the Spring Boot Initializr page to download a pre-configured Spring Boot project, which you can then import into your IDE.

<https://start.spring.io/>



# Entry Point: @SpringBootApplication



This is all we need to have a running Boot application.

@SpringBootApplication is a convenience annotation that adds all of the following:

* @Configuration tags the class as a source of bean definitions for the application context.
* @EnableAutoConfiguration tells Spring Boot to start adding beans based on class path settings, other beans, and various property settings.
* Normally you would add @EnableWebMvc for a Spring MVC app, but Spring Boot adds it automatically when it sees **spring-webmvc** on the class path. This flags the application as a web application and activates key behaviors such as setting up a DispatcherServlet.
* @ComponentScan tells Spring to look for other components, configurations, and services in the hello package, allowing it to find the controllers.

# Creating an Executable Jar

To create an executable jar, we need to add the spring-boot-maven-plugin to our pom.xml. To do so, insert the following lines just below the dependencies section:

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

# Get Your Hands Dirty

**https://www.katacoda.com/courses/java/java8-playground**